Quick Reference guide of the STK++ Arrays

Serge Iovleff

March 9, 2017

1 STK++ Arrays and Vectors

1.1 Containers

The containers/arrays you use in order to store and process the data in your application greatly influence the speed and the memory usage of your application. STK++ proposes a large choice of containers/arrays and methods that you can used in conjunction with them.

All the classes are in `namespace STK`.

Data can be encapsulate in one of the following array

```
typedef CArray<Type, SizeRows, SizeCols, Orient> MyCArray;
typedef CArraySquare<Type, Size, Orient> MyCSquare;
typedef CArrayVector<Type, SizeRows, Orient> MyCVector;
typedef CArrayPoint<Type, SizeCols, Orient> MyCPoint;

typedef Array2D<Type> MyArray2D;
typedef Array2DVector<Type> MyVector2D;
typedef Array2DPoint<Type> MyPoint2D;
typedef Array2DUpperTriangular<Type> MyUpperTriangular2D;
typedef Array2DLowerTriangular<Type> MyLowerTriangular2D;
typedef Array2DDiagonal<Type> MyDiagonal2D;
```

- `Type` is the type of the elements like `double`, `float`, etc.
- `Orient` can be either `STK::Arrays::by_col_` (= 1, the default template) or `Arrays::by_row_` (= 0)
- If you don't know the size of your array/vector just use `STK::UnknownSize` (the default template)

> **Note:**
> Only the first template argument is mandatory in `CArray` family.

1.2 Convenience typedef

There exists predefined typedef for the arrays that can be used. Hereafter we give a sample for the CArray class (The type Real is defined as `typedef double Real` by default but it can be overwritten at compilation as `float`)

Listing 1: CArray family

```
// CArray with Real (double by default) data
typedef CArray<Real, UnknownSize, UnknownSize, Arrays::by_col_> CArrayXX;
typedef CArray<Real, UnknownSize, 2, Arrays::by_col_>           CArrayX2;
typedef CArray<Real, UnknownSize, 3, Arrays::by_col_>           CArrayX3;
typedef CArray<Real, 2, UnknownSize, Arrays::by_col_>           CArray2X;
typedef CArray<Real, 3, UnknownSize, Arrays::by_col_by>         CArray3X;
typedef CArray<Real, 2, 2, Arrays::by_col_>                     CArray22;
typedef CArray<Real, 3, 3, Arrays::by_col_>                     CArray33;
// CArray with double data (add d to the type name)
typedef CArray<double, UnknownSize, UnknownSize, Arrays::by_col_>CArrayXXd;
...
// CArray with int data (add i to the typename)
typedef CArray<int, UnknownSize, UnknownSize, Arrays::by_col_>  CArrayXXi;
```

```
...
// Arrays with data stored by rows, the same as above with ByRow added
typedef CArray<Real, UnknownSize, UnknownSize, Arrays::by_row_> CArrayByRowXX;
...
// CArraySquare (like CArray with the same number of rows and columns)
typedef CArraySquare<Real, UnknownSize, Arrays::by_col_>   CSquareX;
typedef CArraySquare<Real, 2, Arrays::by_col_>             CSquare2;
..
typedef CArraySquare<int, 3, Arrays::by_col_>              CSquare3i;
...
typedef CArraySquare<Real, UnknownSize, Arrays::by_row_>   CSquareByRowX;
```

Some of the predefined typedef for the `Array2D` class are given hereafter

Listing 2: Array2D family

```
  typedef Array2D<Real>   ArrayXX;
  typedef Array2D<double> ArrayXXd;
  typedef Array<int>      ArrayXXi;
  typedef Array<float>    ArrayXXff
```

The predefined type STK::Real can be either @c double (the default) or @c float.
For the other kind of containers there exists also predefined types

Listing 3: CArrayVector and CArrayPoint family

```
typedef CArrayVector<Real, UnknownSize, Arrays::by_col_>   CVectorX;
typedef CArrayVector<Real, 2, Arrays::by_col_>             CVector2;
typedef CArrayVector<Real, 3, Arrays::by_col_>             CVector3;
typedef CArrayVector<double, UnknownSize, Arrays::by_col_> CVectorXd;
typedef CArrayVector<double, 2, Arrays::by_col_>           CVector2d;
typedef CArrayVector<double, 3, Arrays::by_col_>           CVector3d;
typedef CArrayVector<int, UnknownSize, Arrays::by_col_>    CVectorXi;
typedef CArrayVector<int, 2, Arrays::by_col_>              CVector2i;
typedef CArrayVector<int, 3, Arrays::by_col_>              CVector3i;

// CArrayPoint
typedef CArrayPoint<Real, UnknownSize, Arrays::by_col_>   CPointX;
typedef CArrayPoint<Real, 2, Arrays::by_col_>             CPoint2;
typedef CArrayPoint<Real, 3, Arrays::by_col_>             CPoint3;
...
typedef CArrayPoint<int, 3, Arrays::by_col_>              CPoint3i;
```

Listing 4: Array2DVector Array2DPoint and Array2DDiagonal

```
  typedef Array2DPoint<Real>    PointX;
  typedef Array2DPoint<double>  PointXd;
  typedef Array2DVector<Real>   VectorX;
  typedef Array2DVector<double> VectorXd;
  typedef Array2DDiagonal<Real> ArrayDiagonalX;
  typedef Array2DDiagonal<int>  ArrayDiagonalXi;
```

1.3 Constant Arrays

It is possible to use constant arrays with all values equal to 1 using the type predefined in the `namespace STK::Const`

```
Const::Identity<Type, Size> i;
Const::Identity<Type> i(10);
Const::Square<Type, Size> s;
Const::Square<Type> s(10);
Const::Vector<Type, Size> v;
Const::Point<Type, Size> p;
Const::Array< Type, SizeRows, SizeCols> a;
Const::Array< Type> a(10, 20);
Const::UpperTriangular< Type, SizeRows, SizeCols> u;
Const::LowerTriangular< Type, SizeRows, SizeCols> l;
```

As usual, only the first template parameter is mandatory.

2 Manipulating Arrays and Vectors

2.1 Basic operations

Constructors are detailed in the document Arrays Constructors. After creation `arrays` can be initialized using comma initializer

```
CVector3  v;     v << 1, 2, 3;
CSquareXX m(3);  m << 1, 2, 3,
                      2, 3, 4,
                      3, 4, 5;
```

and elements can be accessed using the parenthesis (for arrays), the brackets (for vectors) and the `elt` methods

```
v[0] = 3; v.elt(1) = 1; v.at(2) = 2;       // at(.) check index range
m(0,0) = 5; m.elt(1,1) = 1; m.at(2,2) = 3; // at(.) check indexes range
```

The whole array/vector can be initialized using a value, either at the construction or during execution by using

```
v.setZeros()
m.setOnes();
Array2D<int> a(3, 3, 1); // arrays of size (3,3) with all the elements equal to 1
a.setValue(2);
```

2.2 Arrays and vectors getters

For all the containers it is possible to get the status (reference or array owning its data) the range, the beginning, the end and the size using

```
CArrayXX m(3,4);
bool mf = m.isRef(); // mf is false

Range mc= m.cols();
int mbc= m.beginCols(), mec= m.endCols(), msc= m.sizeCols();

Range mr= m.rows();
int mbr= m.beginRows(), mer= m.endRows(), msr= m.sizeRows();

CArrayVectorX v(m.col(0), true); // v is a reference on the column 0 of m
bool vf = v.isRef(); // vf is true

Range vr= m.range();
int vb= m.begin(), ve= m.end(), vs= m.size();
```

For the Array2D family of container, it is also possible to get informations about the allocated memory of the containers. It can be interested in order to known if the container will have to reallocate the memory if you try to resize it.

```
ArrayXX m(3,4);
m.availableCols();                  // number of available columns
Array1D<int> a = m.availableRows(); // vector with the number of available rows of each column

m.capacityCol(1);                   // capacity of the column 1
Array1D< Range > r = m.rangeCols () // vector with used range of each columns (think to triangular
     matrices)
```

The Array2D family of container allocate a small amount of supplementary memory so that in case of a `resize`, it is possible to expand the container without data transfer.

2.3 Arrays and vectors visitors and appliers

Visitors and appliers visit/are applied to an array or vector (or expression) and are automatically unrolled if the array is of fixed (small) size.

Listing 5: Visitors

```
m.count();
m.any();
```

```
m.all();
m.minElt(i,j);      // can be v.minElt(i) or m.minElt()
m.maxElt(i,j);      // can be v.maxElt(i) or m.maxElt()
m.minEltSafe(i,j); // can be v.minEltSafe(i) or m.minEltSafe()
m.maxEltSafe(i,j); // can be v.maxEltSafe(i) or m.maxEltSafe()
m.sum(); m.sumSafe();
m.mean(); m.meanSafe();
```

Listing 6: Appliers

```
m.randUnif();  // fill m with uniform random numbers between 0 and 1
m.randGauss(); // fill m with standardized Gaussian random numbers
m.setOnes();   // fill m with value 1
m.setValues(2);// fill m with value 2
m.setZeros();  // fill m with value 0
Law::Gamma law(1,2); // create a Gamma distribution
m.rand( law);        // fill m with gamma(1,2) random numbers
```

The next methods use visitors in order to compute the result, eventually safely

```
m.norm();  m.normSafe();
m.norm2(); m.norm2Safe();
m.normInf();
m.variance(); m.varianceSafe();
// variance with fixed mean
m.variance(mean); m.varianceSafe(mean);
// weighted visitors
m.wsum(weights);  m.wsumSafe(weights);
m.wnorm(weights);  m.wnormSafe(weights);
m.wnorm2(weights); m.wnorm2Safe(weights);
m.wmean(weights);  m.wmeanSafe(weights);
m.wvariance(weights); m.wvarianceSafe(weights);
m.wvariance(mean, weights); m.wvarianceSafe(mean, weights);
```

3 Functors on arrays/vectors and expressions

All the functors applied on arrays are currently in the namespace STK::Stat. If there is the possibility of missing/NaN values add the word Safe to the name of the functor. If the mean by row is needed, just add ByRow to the name of the functor. If you want a safe computation by row add SafeByRow.

All functors applied on an array by column will return by value an STK::Array2DPoint and a number if it is applied on a vector or a point.

All functors applied on an array by row will return by value an STK::Array2DVector and a number if it is applied on a vector or a point.

Listing 7: Functors by column

```
Stat::min(a); Stat::minSafe(a); Stat::min(a, w); Stat::minSafe(a, w);
Stat::max(a); Stat::maxSafe(a); Stat::max(a, w); Stat::maxSafe(a, w);
Stat::sum(a); Stat::sumSafe(a); Stat::sum(a, w); Stat::sumSafe(a, w);
Stat::mean(a); Stat::meanSafe(a); Stat::mean(a, w); Stat::meanSafe(a, w);
// could also be (safe versions)
Stat::minByCol(a); Stat::minSafeByCol(a); Stat::minByCol(a, w); Stat::minSafeByCol(a, w);
// .. etc
// unbiased variance (division by n-1) when unbiased is true, false is the default
Stat::variance(a, false); Stat::varianceSafe(a, false);
Stat::variance(a, w, false); Stat::varianceSafe(a, w, false);
// fixed mean. Must be a vector/point for arrays and a number for vectors/points
// unbiased (false in examples below) has to be given
Stat::varianceWithFixedMean(a, mean, false);
Stat::varianceWithFixedMean(a, w, mean, false);
Stat::varianceWithFixedMeanSafe(a, mean, false);
Stat::varianceWithFixedMeanSafe(a, w, mean, false);
```

Listing 8: Functors by row

```
Stat::minByRow(a); Stat::minSafeByRow(a); Stat::minByRow(a, w); Stat::minSafeByRow(a, w);
Stat::maxByRow(a); Stat::maxSafeByRow(a); Stat::maxByRow(a, w); Stat::maxSafeByRow(a, w);
Stat::sumByRow(a); Stat::sumSafeByRow(a); Stat::sumByRow(a, w); Stat::sumSafeByRow(a, w);
Stat::meanByRow(a); Stat::meanSafeByRow(a); Stat::meanByRow(a, w); Stat::meanSafeByRow(a, w);
// unbiased variance (division by n-1) when unbiased is true, false is the default
Stat::varianceByRow(a, false); Stat::varianceSafeByRow(a, false);
Stat::varianceByRow(a, w, false); Stat::varianceSafeByRow(a, w, false);
// fixed mean. Must be a vector/point for arrays and a number for vectors/points
// unbiased (false in examples below) has to be given
Stat::varianceWithFixedMeanByRow(a, mean, false);
Stat::varianceWithFixedMeanByRow(a, w, mean, false);
Stat::varianceWithFixedMeanSafeByRow(a, mean, false);
Stat::varianceWithFixedMeanSafeByRow(a, w, mean, false);
```

4 Arithmetic Operations on arrays/vectors and expressions

Available operations on arrays/vectors and expressions are summarized in the next table. Many operations are similar to the operations furnished by the Eigen library .

Listing 9: add subtract divide multiply arrays element by element"

```
  m= m1+m2; m+= m1;
  m= m1-m2; m-= m1;
  m= m1/m2; m/= m1;
  m= m1.prod(m2); // don't use m1*m2 if you want a product element by element
```

Listing 10: add subtract divide multiply a number

```
  Real s;
  m= m1+s; m= s+m1; m+= s;
  m= m1-s; m= s-m1; m-= s;
  m= m1/s; m= s/m1; m/= s;
  m= m1*s; m= s*m1; m*= s;
```

Listing 11: matrix by matrix/vector products

```
  ArrayXX m2(5,4), m1(4,5), m; PointX p2(5), p1(4), p; VectorX v2(4), v1(5), v;
  Real s = p2v1; // dot product
  v= m2*v2; v= m1p2.transpose(); // get a vector
  p= p2*m2; p= v1.transpose()*m2;   // get a point (row-vector)
  m= m2*m1; m = m1.transpose()m; // matrix multiplication
  m= m2.prod(m1.transpose());       // product element by element
  m2*= m1;                  // m2 will be resized and filled with m2*m1 product
```

Listing 12: Comparisons operators

```
Real s;
// count for each columns the number of true comparisons
count(m1 < m2);  count(m1 > m2);  count(m1 < s);  count(m1 > s);
count(m1 <= m2); count(m1 >= m2); count(m1 <= s); count(m1 >= s);
count(m1 == m2); count(m1 != m2); count(m1 == s); count(m1 != s);
```

Listing 13: Miscellaneous functions

```
  m.isNa();       // boolean expression with true if m(i,j) is a NA value
  m.isFinite();   // boolean expression with true if m(i,j) is a finite value
  m.isInfinite(); // boolean expression with true if m(i,j) is an infinite value
  m.min(m2);      // Type expression with min(m(i,j), m2(i,j))
  m.max(m2);      // Type expression with max(m(i,j), m2(i,j))
  m.prod(m2);     // Type expression with m(i,j)*m2(i,j)
```

```
m.neg();         // Type expression with !m(i,j)
m.abs();         // Type expression with abs(m(i,j))
m.sqrt();        // Type expression with sqrt(m(i,j))
m.log();         // Type expression with log(m(i,j))
m.exp();         // Type expression with exp(m(i,j))
m.pow(number);   // Type expression with m(i,j)^number
m.square();      // Type expression with m(i,j)*m(i,j)
m.cube();        // Type expression with m(i,j)*m(i,j)*m(i,j)
m.inverse();     // Type expression with 1./m(i,j)
m.sin();         // Type expression with sin(m(i,j))
m.cos();         // Type expression with cos(m(i,j))
m.tan();         // Type expression with tan(m(i,j))
m.asin();        // Type expression with asin(m(i,j))
m.acos();        // Type expression with acos(m(i,j))
m.cast<OtherType>(); // OtherType expression with static_cast<OtherType>(m(i,j)))
```